Vol.2018-HPC-167 No.5
2018/12/17

逐次プログラムからマルチコア・マルチノード並列処理への変換を容易にするディレクティブベース API SMint

阪口 裕梧[†1]，西矢 和生[†1]，緑川 博子[†1]

概要： 筆者らが構築するソフトウエア分散共有メモリ mSMS は，クラスタにおいて大規模グローバルデータを提供し，効率的なマルチコア・マルチノード並列処理と高生産性なプログラミング環境を実現する．本報告では，従来の SMS ライブラリ関数と分散マッピング機能付き shared 大域データ配列宣言によるプログラミング手法に加え，逐次プログラムから容易にマルチノード並列プログラムに変換が可能なディレクティブベース API（SMint）を提案する．SMint は，典型的なループ文をマルチノード並列に極めて容易に変換する．さらに，並列処理部分で予め必要なデータがわかっている場合には，並列処理セクション開始時に遠隔ノードに割り付けられたデータ領域を一括転送してローカルノードにキャッシュし，並列処理セクション終了時に，個々のデータに応じたキャッシュの更新，廃棄を行い，並列計算効率を高める．これにより，典型的なデータ並列処理を，クラスタ向けのプログラムに容易に変換し，高性能な処理を実現する．

キーワード：PGAS，インクリメンタルプログラミング，ディレクティブベース，API，共有メモリプログラミングモデル，マルチスレッド，マルチノードプログラミング，クラスタ，ソフトウエア分散共有メモリ

1. はじめに

高性能計算応用においては，高速ネットワークで結ばれた複数計算ノードとノード内マルチコア(あるいは GPU などのアクセラレータ）を有効利用するため，MPI＋X（OpenMP[8]，OpenACC[10] など）によるプログラミング手法が広く用いられている．また，分散メモリモデルである MPI によるプログラムの書きにくさ，プログラム開発の生産性の低さを改善するために，PGAS（Partitioned Global Address Space）モデルで総称される様々な言語や API などが提案されている[1][12][15]． 現在，PGAS として，大域的データのグローバルビューを提供しながらデータの所在に考慮した様々なシステムが登場し，一定レベル以上の性能と MPI よりも高いプログラム生産性，あるいは，新しい並列実行モデルの提供，もしくは特定応用向けに高性能化など，様々な方向への研究が行われている．このため，PGAS とは，事実上，多種多様な実行モデルとシステム，言語を包含する名称になっている．多くの PGAS では，MPI と異なり，大域データ配列宣言や大域インデックスを利用できることでグローバルビューとしているが，各ノードプロセスがアクセスできる遠隔データの範囲に制限があったり，遠隔データ利用前に事前にアクセス領域の宣言が必要である場合もある．遠隔データアクセス API においても，データの所在（ノード番号など）を付してデータにアクセスする(coaray 記述)[15]，あるいは MPI 片側通信(get/put)のような API などを必要とする場合もある．多くの場合，実行プロセスに対し，真の同一共有メモリアドレス空間を提供しておらず，特定のコンパイラを介して，上記の API を下層通信実装の関数（MPI など）に変換している．すなわち，全ノードプロセスが，同一のアドレス空間を共有して，C 言語のポインタ変数やアドレスを使ったアクセスを可能とするシステムはほとんどない．また，計算ノード識別のための API がなく，全ノードが同じ処理を行う collective 型の処理を基本とする PGAS もあり，ノード毎に異なる処理をする非対称な処理応用に向かない場合もある．

筆者らは，古典的ページベースのソフトウエア分散共有メモリ（SDSM）[2]に，ノード間マルチノードプログラミングを効率的に行うことのできるプログラミング環境を実現した[3]．さらに，ノード内マルチコアプログラミングとノード間マルチノードプログラミングをシームレスに行うために，以下の特徴を備えた mSMS を構築した[5]．

- 動的に生成・消滅する複数のユーザスレッドからの非同期・範囲無制限の遠隔データアクセスに対応．（マルチノード＋pthread，OpenMP，OpenACC，Cuda）
- 計算ノード間のページ送受信を高速化するマルチスレッドによる送受信通信機構
- 予めデータアクセス範囲が予測可能な時，計算（アクセス）前に大域データをまとめてローカルにフェッチする preload 機構

mSMS[5]では，マルチノードマルチスレッドシステムにおいて，各ノードの物理メモリサイズとノード数に応じた大規模大域データを定義でき，大域データのアクセスに制限はない．各ノードでのマルチコア並列処理は，OpenMP や pthread などにより記述できる．大域データは，任意のノードへ割り付けが可能で，ローカルメモリアクセスを高めるように，ユーザがデータ割り付けを自由に決めることができる．また．mSMS は，現在，MPI を下層通信に用い，ユーザレベルソフトウエアで実装されているため，管理者権限は不要で誰でも容易に利用可能である．

†1 成蹊大学　Seikei University.

筆者らは，大規模クラスタシステム Tsubame3.0 [6] において，全体ノード数（540）の 1/3 にあたる 180 ノードを利用して，典型的な計算カーネルである 3 次元配列のステンシル計算のマルチノードマルチスレッド並列処理を行い，mSMS の稼働・性能の調査を行った[7]．各ノードに約 130GB のデータを分散配置して，全体として 23TiB の大域共有データを定義し，約 30TiB の仮想アドレス空間を持つプロセスを各ノードで稼働させて，大規模な共有メモリデータプログラミング環境を実現した．この結果，ステンシル計算のような典型的データ並列処理の場合，MPI による実装とほぼ，同等性能を達成している。この実験では，mSMS が提供する SMS ライブラリ関数と，マルチノードにおけるグローバル共有データ配列宣言 shared を利用し，従来の１ノードにおける C プログラム開発と同等な生産性のよいプログラミング環境で処理を実現した．

これまでの２つプログラミング環境，(1)SMS ライブラリ関数を使った C プログラムと，(2)マルチノードにおけるグローバル共有データ配列宣言 shared が利用可能な MpC プログラム[3]に加え，現在,(3)#pragma 利用によるディレクティブベースの API，SMint を構築中で，本報告では，これについて紹介する．

SMint は，現在広く用いられているマルチコア並列処理のための OpenMP，OpenACC と同様に，逐次プログラムに #pragma 文を加えることにより，典型的なマルチノード並列処理（for 文の並列化，データ分割並列処理）を，極めて容易に行うことができる．さらに，並列処理部分で予め必要なデータがわかっている場合には，マルチノード並列セクション開始時に遠隔ノードに割り付けられたデータ領域を一括転送してローカルノードにキャッシュし，並列セクション終了時には，個々のデータに応じたキャッシュの更新，廃棄を行い，データ転送を効率化する．これにより，典型的なデータ並列処理を，クラスタ向けのプログラムに容易に変換し，高性能な処理を実現する．

2. mSMS における並列プログラミング

2.1 mSMS における大域データの扱い

mSMS ではマルチノードで共有する大域データと，各ノードのローカルデータに大別できるが，UPC[12][13]のように，大域データ（shared）向けポインタと局所データ用のポインタを区別することはしていない．従来の C のポインタやアドレスの利用はそのまま大域データにも行うことができる．したがって，sms_alloc を malloc に変更すれば，ローカルメモリのみを利用する通常プログラムになる．

ノードが共有する大域データに関しては，緩和型メモリ一貫性モデルを採用しているため，データ一貫性同期（sms_barrier）を行うまでは，あるノードのデータ変更は他のノードには見えない．データの一貫性制御の内部実装は，ページベースソフトウエア分散メモリで広く用いられるページの差分 diff を用いており，diff を該当ページのホームノードに転送し，ホームノードでは集まった diff を集約して該当ページを最新にする[2].

大域データのアドレス空間領域（SMS データ領域）と，ローカルノードデータのアドレス空間領域は分離しているが，プログラムにおけるデータ操作に差異はない．通常，大きな shared データは，その一部のみがローカルメモリにあり，他の shared データ領域は複数の遠隔ノードに分散されてマップされる．すなわち，利用可能ノード数が多ければ，より巨大な大域データを扱うことができる．しかし，ローカルノードにキャッシュできる遠隔データの量は，ローカル物理メモリサイズにより制限を受けるため，定期的なデータ一貫性同期を行い，キャッシュされたデータを反映，または廃棄し，キャッシュが巨大化するのを防ぐ必要がある．

利用する計算システムに応じて，mSMS が使用できる１ノードあたりのローカル物理メモリサイズを設定ファイルに記述する．また，応用のメモリアクセスや処理パターンに応じて，ローカルノードに割り付ける大域データ領域とキャッシュ領域のサイズの比を設定することができる．たとえば，応用処理で，データのアクセス局所性が高い場合（ほとんどのアクセスがローカルノードに割り付けられた大域データの場合）には，キャッシュ領域を小さくし，より大きい大域データ領域をローカルに割り付けて巨大データを処理することもできる．反対に，キャッシュ領域サイズを大きくすると，データ一貫性同期を少なくできる場合もある．

大域データは，ローカルデータと同様にアクセスできるが，１ノードの計算において，キャッシュヒットを高めるアルゴリズムが重要であるのと同様に，複数ノード処理において，ローカルノードメモリアクセスをいかに高めるかが性能向上の鍵となる．

2.2 SMS ライブラリ関数利用による C プログラム

mSMS における基本プログラミングは，表 1 に示すような SMS ライブラリ関数を用いて C で作成するものである．MPI と同様に，各ノードの rank 番号とプロセス数に対応する sms_rank，sms_nprocs 定数が利用できるので，ノード毎に異なる処理をさせることもできる．図 1 に行列ベクトル積のプログラム例を示す．マルチノード上で共有される大域データは，sms_alloc あるいは，sms_mapalloc を用いて，動的に確保する．sms_alloc は一つの指定ノードに指定サイズの大域データを割り付ける．sms_mapalloc は，主に大

域データ配列の分散マッピング用で，配列次元毎に分割数を指定し，指定した複数の割り付けノードに，サイクリックにデータを分散マッピングする[3]（図２参照）．次節のMpC プログラミングにおける shared データの配列宣言は，この sms_mapalloc に変換されている．

2.3 大域データ型配列宣言による MpC プログラム

第二のプログラミング手法は，図３に示すようなMpC プログラムによるもので，データ分散マッピング指定付き多次元配列宣言を利用することができる．MpC は，C に対し最小限の拡張を施したもので[3]，新たに導入した大域データ型 shared による配列宣言を可能にしている．数値計算などで用いることの多い配列宣言が記述しやすい．MpC プログラムは，MpC トランスレータによって，図１と同等なCプログラムに自動変換され，最終的には汎用Cコンパイラにより実行プログラムが作られる．もともと MpC は，MpC トランスレータに与える指示によって，3 種の分散共有メモリ（TreadMark，JIAJIA，SMS）と pthread プログラムのいずれかに変換できるように設計されていたが[3]，ここでは，mSMS に変換されるようにしている．

いずれのプログラムでも，スレッド実装された既存の汎用数学ライブラリ関数や，OpenMP, OpenACC，pthread 関数などを，各ノード処理部分にそのまま使用できる．

表１　mSMS 定数とmSMS 関数（一部）

定数	概要
sms_rank	smsプロセス番号（ノード番号，0 - sms_nprocs-1）
sms_nprocs	実行プログラム中のsmsプロセス（ノード）数
関数名（引数略）	概要
sms_startup ( )	SMSの利用開始
sms_shutdown ( )	SMSの利用終了
sms_error( )	エラー時のSMSの利用終了
sms_alloc( )	sharedデータの確保（指定ノードに割り付け）
sms_mapalloc( )	sharedデータの分散マッピング割り付け
sms_barrier( )	実行同期，データー貫性同期（データ更新）
sms_sync_drop()	実行同期，データー貫性同期（キャッシュ廃棄）
sms_sync( )	実行同期
sms_preload( )	sharedデータ領域の一括転送（キャッシング）
sms_overload( )	sharedデータ領域の一括転送（キャッシング）上書き
sms_preload_array( )	shared配列　部分領域の一括転送（キャッシング）
sms_overload_array( )	shared配列　部分領域の一括転送（キャッシング）上書き
sms_get( )	sharedデータ領域をローカルデータ領域に一括転送
sms_get_array( )	shared配列　部分領域をローカル配列に一括転送
sms_put( )	ローカルデータ領域をsharedデータ領域に一括転送
sms_put_array( )	ローカル配列を　shared配列　部分領域に一括転送
sms_fread( )	sharedデータへのファイルリード
sms_fwrite( )	sharedデータからのファイルライト

```
#include <sms.h>                  //  SMSライブラリ関数利用によるC プログラム
#define N …
int main(int argc, char *argv[] )
{   int size, st, ed;              // 各ノードの担当領域
    double *vec1, *vec2;  // 1次元配列 vec1[N], vec2[N] のためのポインタ
    double (*array) [N];  // 2次元配列 array[N][N] のためのポインタ
    int dim[3]={N, N,-1}, div[3]={1, 1,-1}; // dim: 配列サイズ, div:分散マップ分割数

    sms_startup(&argc, &argv);

    vec1 = (double*)sms_alloc(sizeof(double), N, 0); // vec1[N] node0に割り付け
    vec2 = (double*)sms_alloc(sizeof(double), N, 1); // vec2[N] node1に割り付け
    div[0]=sms_nprocs;          //   arrayをバンド分割，全ノードに分散マッピング
    array= (double(*)[N]) sms_mapalloc(dim, div, sizeof(double), 0, sms_nprocs );

    size=N/sms_nprocs;
    st=size * sms_rank; ed=size * (sms_rank+1); //各ノード担当領域
  #pragma omp parallel for          //  各ノードでは，マルチスレッド実行
    for( i=st; i<ed; i++) {              // 全ノードで for(i=0; i<N; i++) を並列実行
       for(k=0; k<N; k++)    vec2[i]= array[i][k] * vec1[k];     // 行列ベクトル積
    }
    sms_barrier();
    sms_shutdown();
}
```

図１　SMS ライブラリ関数利用によるCプログラム

shared 型名 変数名[sm]..[si]..[so] ::[dm]..[di]..[do] (st,n)

shared Cの記憶クラス指定子として組み込み(shared, global, local)

[dm]..[di]..[do]　　各次元の分割数

st 割付開始プロセスID（省略時は0,, または任意プロセス）
n 割付に使用するプロセス数（省略時は全プロセス個数）

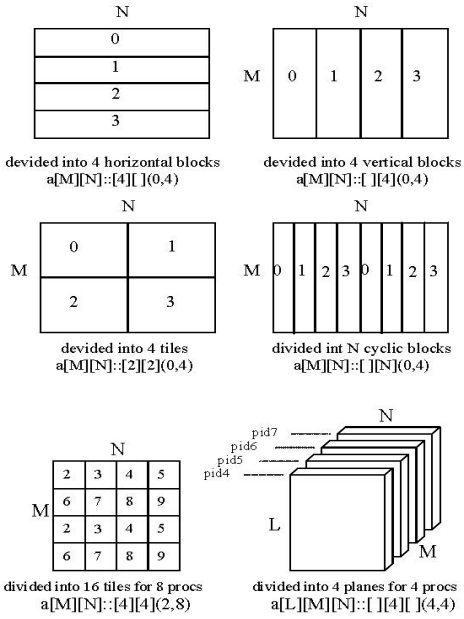

図２ MpC における shared データ分散マッピング記述

```
#include <mpc.h>      // MpC プログラム（sharedデータ宣言利用）
#define N …
shared double vec1[N] ::[1](0,1) ;     // node0にマップ
shared double vec2[N] ::[1](1,1) ;     // node1にマップ
shared double array[N][N] ::[NPROCS][](0,NPROCS) ;
                                       //全ノードに分散マップ
int main(int argc, char *argv[] )
{   int size, st, ed;                  // 各ノードの担当領域

    mpc_init(&argc, &argv);

    size=N/NPROCS;
    st=size * MYPID; ed=size * (MYPID+1); //各ノード担当領域
  #pragma omp parallel for   //   各ノードでは，マルチスレッド実行
    for( i=st; i<ed; i++){      // 全ノードで for(i=0; i<N; i++) を並列実行
       for(k=0; k<N; k++)
            vec2[i]= array[i][k] * vec1[k];     // 行列ベクトル積
    }
    mpc_barrier();
    mpc_exit();
}
```

図３　MpC 利用によるプログラム

3. ディレクティブベース並列処理 API

本報告で提案する SMint は，ディレクティブベースの API で，ループ文をマルチノード並列にする機能[4]に加え，マルチノード並列セクションの前後での遠隔データの一括転送などの指示句を加えることにより，より効率的なプログラム記述を可能とする．

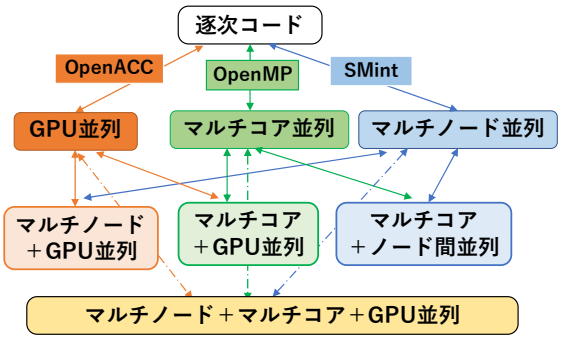

図４ 階層的インクリメンタルプログラミング

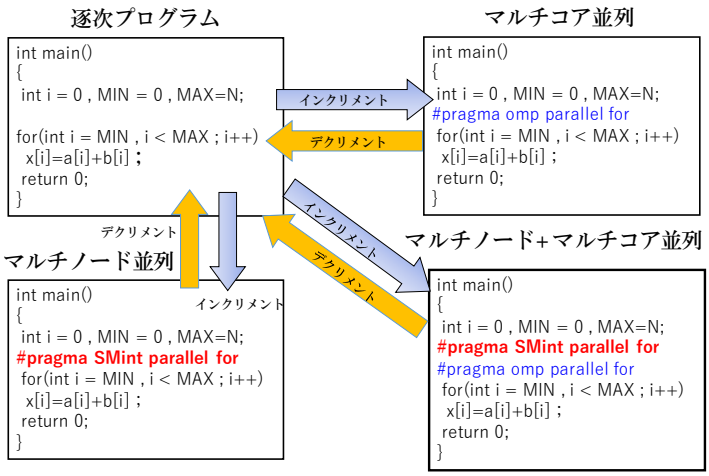

図５ OpenMP と SMint との組み合わせ例

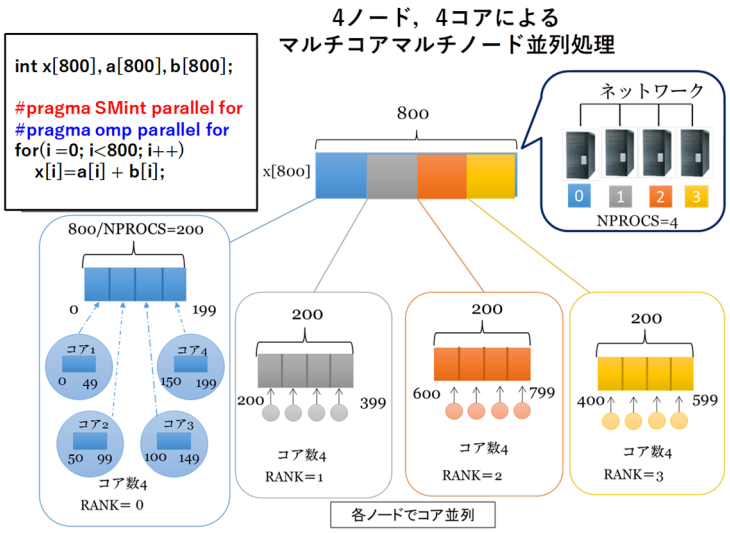

図６ マルチノードマルチスレッド並列処理のイメージ

3.1 SMint インクリメンタルプログラミング

#pragma を利用する並列処理 API としては，マルチコア向けの OpenMP[8][9]や，GPU 向けの OpenACC [10][11]，マルチノード向け PGAS 言語 XcalableMP[15][16]などがある．OpenMP，OpenACC では，逐次プログラムに#pragma を加えることで容易に並列処理を行うことができ，#pragma を無視することで逐次プログラムとしても実行できるというインクリメンタルプログラミングのモデルを実現している．特に，ループ文による典型的処理をデータ並列処理プログラムに変換する OpenMP の #pragma omp parallel for は，広く用いられている．

SMint においても，逐次プログラムに#pragma SMint を加えることで，クラスタにおけるマルチノード並列処理を可能とする．さらに，図４に示すように，OpenMP や OpenACC などの他の並列化 pragma と自由に組み合わせて利用することも，それぞれ単体で利用することも可能で，柔軟性の高い並列プログラミング環境を提供する．図５は，OpenMP と SMint の組み合わせ例の概要を示している．また，図６にクラスタにおけるマルチノードマルチコア並列処理による配列の処理のイメージを示す．

3.2 SMint トランスレータ

SMint は，図７のように，前述の SMS 関数のみによる C プログラム(図１)と MpC プログラム（図２）の上層に位置する API と言える．SMint トランスレータは， #pragma 文の変換，shared 配列の変換（MpC トランスレータと同等の変換）後，最下層の SMS 関数利用 C プログラムに変換し，最終的に汎用 C コンパイラを用いて実行ファイルを作成する．現在，C コンパイラには gcc を使用いている．

3.3 SMint ループ並列処理と shared 配列

SMint では，for 文の前に#pragma を挿入すること

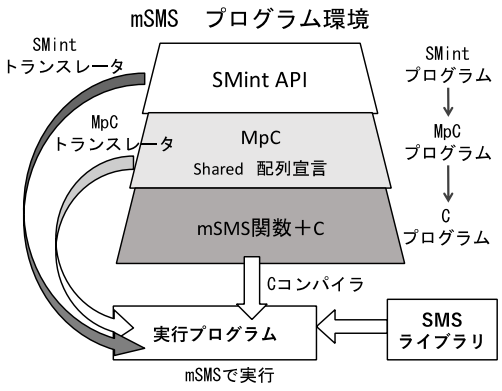

図７ mSMS におけるプログラム開発

により，OpenMP と同様に，自動的に for 文の処理範囲を分割して，マルチノード並列処理を行うことができる．図 8 に図 1，図 3 と同等のプログラムを SMint で記述した場合のプログラム例を示す．MpC における shared データ配列宣言の MpC 特有の分散マッピング指示部分は，#pragma として分離している．インクリメンタルプログラミングとして，#pragma 文を無視すると，逐次プログラムに変換できるようにするためである．また，図 3 にあったマルチノードの実行・データ一貫性同期 mpc_barrier() (MpC トランスレータで sms_barrier()に変換される) は，SMint トランスレータにより，マルチノード並列セクション（この場合は for 文）の終わりに自動挿入されるため，ユーザによる記述は不要になる．図 9 に SMint トランスレータのループ並列部の変換例を示す．

3.4 並列処理部におけるデータローカライズ指示

#pragma を用いる OpenMP[8]では，マルチスレッド並列処理セクションの前後において，プログラム上のデータ名は維持しつつ，指定した大域データをスレッドローカルデータに変換して（firstprivate, private, lastprivate など），スレッド間で独立に並列処理を効率的に行えるようにしている．同様に，OpenACC[10]においても，ホストで定義したデータ名は維持しつつ，GPU カーネル処理部の前後に，ホストメモリの配列データの一部などを GPU メモリにコピーするなどの指示（copyin, copy, copyout, create など）

```
#include <smint.h>        // #pragma SMint 利用プログラム
#define N …
#pragma SMint shared ::[1](0,1) ;     // node0にマップ
double vec1[N] ;
#pragma SMint shared ::[](1,1) ;      // node1にマップ ,省略記述
double vec2[N] ;
#pragma SMint shared ::[NPROCS][](0,NPROCS) ;//全ノードに分散マップ
double array[N][N] ;

int main(int argc, char *argv[] )
{   int size, st, ed;              // 各ノードの担当領域

  #pragma SMint parallel for  // 全ノードで並列実行
  #pragma omp parallel for    //各ノードでマルチスレッド実行
   for( i=0; i<N; i++) {
     for(k=0; k<N; k++)
          vec2[i]= array[i][k] * vec1[k];    // 行列ベクトル積
   }
   // バリア同期は自動挿入
}
```

図 8　SMint 利用によるプログラム

```
#pragma SMint parallel for
#pragma omp parallel for
for( i=0; i<N; i++) {
   for(k=0; k<N; k++)
     vec2[i]= array[i][k] * vec1[k];
}
```

SMintプログラム

```
{
int __start,__end,__count,__rcount;
__count = ((N)-(0))/NPROCS;
__rcount = ((N)-(0))%NPROCS;
if(__rcount > MYPID) __count++;
__start = (0)+MYPID*(((N)-(0))/NPROCS)+
        ((__rcount>MYPID)?  MYPID:__rcount );
__end =__start+__count;

 #pragma omp parallel for
 for( I = __start ; I < __end; i++) {
   for(k=0; k<N; k++)
     vec2[i]= array[i][k] * vec1[k];
 }
 sms_barrier();
}
```

MpCプログラム

図 9　SMint におけるマルチノード　ループ並列の変換例

```
#pragma omp parallel for private( j, k )
 for( i=0; i<N; i++ )
   for( k=0; k<N; k++ )
     for( j=0; j<N; j++ )
       c[i][j]+=a[i][k]*b[k][j];
```

OpenMP：行列積

```
void calc(int n, const float *a, const float *b, float c, float *d)
 {
#pragma acc kernels present(a, b, d)
#pragma acc loop independent
  for (int i=0; ii<n; i++) {
       d[i] = a[i] + c*b[i];
   }
}
int main()
{
 …
#pragma acc data copyin(a[0:n], b[0:n]) copyout(d[0:n])
  {
     calc(n, a, b, c, d);
  }
   …
}
```

OpenACC：ベクトル和　計算

図 10　OpenMP と OpenACC のデータローカライズ例

表 2　データローカライズ指定の比較

OpenMP	OpenACC	SMint
	copy	copy
firstprivate	copyin	copyin / scopyin
private	create	create / screate
lastprivate	copyout	copyout

を加える．図 10 にそれぞれの使用例を示す．両者のメモリ構成やハードウエアは全く異なるものであるが，並列処理部の前後でデータのローカライズを行うという観点ではプログラム上，類似点がある．

mSMS では，どのノードからいつでも shared データにアクセスすることはできるが，一方で，処理途中で遠隔データを 1 ページずつフェッチすると計算が一時中断され効率が悪い．このため，予めアクセスするデータ領域がわかっている場合には，表 1 の sms_preload 関数や sms_overload 関数により，ローカルノードに処理対象の遠隔データ領域を計算開始前にまとめて転送しキャッシュしてから計算を開始しするほうが，一般には効率がよい．sms_preload は，従来のページフォルト時の 1 ページ毎の遠隔ページフェッチを連続領域に対し，一度に転送できるようにしたものである．これらの関数は，先の大規模クラスタ実験[7]においても効果が確かめられている．

そこで，SMint では，ループ並列処理セクションの前後に OpenACC と同様な，大域データ shared 配列の部分配列をローカライズ指定できる API を用意している．SMint における大域データのローカライズとは，実際にはマルチノード上に分散マッピングされた大域配列データのうち，ローカルノードにない部分のデータを，並列処理を行う前にあらかじめローカルノードにキャッシュしておくことに該当する．表 2 は，

OpenMP, OpenACC, SMint におけるデータローカライズのための指示句の比較を示す．OpenMP4.5 以降の仕様 [9]では，汎用アクセラレータ向け API として target data 構文によりデータ配列のマップ指示句も定められているが，表 2 では，現在, OpenMP で広く用いられているスカラー変数のローカライズ指定を示している. OpenACC における指示句は，ホストメモリデータと GPU メモリデータ間のデータ転送に該当する．

3.5 SMint　並列処理部のデータローカライズ指示

SMint は，表２の右端に示すように，OpenACC と同等な指示名を採用している．

copyin は，並列セクション開始時に該当データをローカルノードに一括キャッシュし，並列セクション終了時にキャッシュしたデータを廃棄する（discard）.

copyout は，並列セクション開始時に，ローカルノードにおいて該当大域データ領域を読み書き可能とするが，遠隔ノードからの実際のデータ転送は行わず値は不定のままとしておく．並列セクション終了時には，キャッシュしたデータを遠隔ノードに送って値の更新を行う(reflect). copyout は，並列セクションで，該当領域を読まずに，計算結果などを書き込む場合に使用される．終了時のデータ更新は，指定領域を構成する各ページの保持ノードに対し，連続領域の一括変更とし，従来の diff ベースの更新は行わない．

copy は，copyin と同様，並列セクション開始時に該当データを一括キャッシュし，並列セクション終了時には，copyout と同様に, キャッシュしたデータを遠隔ノードに送って値の更新を行う．

create は，並列セクション開始時に，copyout と同様に，ローカルノードにおいて該当データ領域を読み書き可能とするが，遠隔ノードからの実際のデータ転送は行わない．並列セクション終了時には，copyin と同様に，キャッシュしたデータを廃棄する（discard）.

scopyin と screate は，それぞれ袖領域の copyin と create に対応しており，次元毎に袖の幅を指定すると，その次元の隣接領域から指定幅の領域を，読み書き可能とし，scopyin の場合には遠隔からデータをキャッシュする. 図 11 に示すように，袖領域サイズが非対称な場合も指定可能である．

図 12 に，並列セクションにおける copyin 指示のプログラム例を示す．SMint プログラムのマルチノード並列セクションの開始時に，ノード 0 に割り付けられた vec1[N]配列をそれぞれのローカルノードにキャッシュすることを指示している．図中に示すように，行列 array[N][N]とベクトル vec2[N]は，あらかじめ全ノードに分散マッピングされているため，並列処理部ではローカルメモリアクセスしか発生しない．copyin(vec1[])とすることにより，vec1[N]配列すべてがローカルにキャッシュされ，並列処理実行中に，遠隔ノード（この例ではノード０）からデータ転送を行う必要がない．

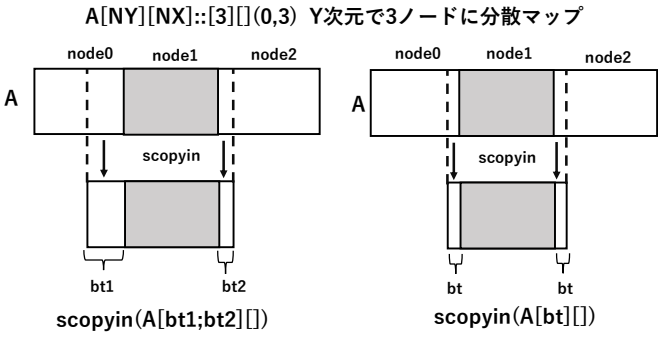

図 11　袖領域のデータローカライズ scopyin

```
#include <smint.h>          // #pragma SMint 利用プログラム
#define N …
#pragma SMint shared ::[](0,1) ;      // node0にマップ
double vec1[N] ;
#pragma SMint shared ::[NPROCS](0, NPROCS ) ; //全ノードに分散マップ
double vec2[N] ;
#pragma SMint shared ::[NPROCS][](0,NPROCS) ; //全ノードに分散マップ
double array[N][N] ;

int main(int argc, char *argv[] )
{   int size, st, ed;

  //並列実行開始前に全ノードで, vec1(node0にマップ) をローカルにキャッシュ
  #pragma SMint parallel for  copyin (vec1[ ])    //全ノードで並列ループ実行
  #pragma omp parallel for
  for( i=0; i<N; i++) {
     for(k=0; k<N; k++)
          vec2[i]= array[i][k] * vec1[k];     // 行列ベクトル積
  }
}
```

vec2[N] = array[N][N] * vec1[N]

vec2[N]	array[N][N]	vec1[N]
n0	n0	n0
n1	n1	
n2	n2	
n3	n3	

4node利用時の分散マップ

図 12　SMint における copyin 利用プログラム例

これらの機能は，表３に示す新たに導入した部分配列向けの sms 関数を，並列セクション開始時と終了時にそれぞれ呼び出すことにより実現する．これらの関数は，データローカライズ指示で指定された配列毎に呼ばれる．

既存の mSMS のデータ一貫性モデルでは，すべての大域変数を，キャッシュを反映してデータ更新をするか（sms_barrier），キャッシュを廃棄してデータ変更しないか(sms_sync_drop)を指示するものであったが，SMint では，個々の指定領域（部分配列について）について，異なったデータ更新方式を適用し，あらかじめアクセス領域のわかっている場合の一括データ転送だけでなく，開始時や終了時の無駄なデータ転送も省く．

図 13 に，配列 A,B,C の copyin，create，copyout を並列セクションに付加した場合に，どのように変換されるかを示す．

表 3 SMint データローカライズ指定と利用関数

SMint データローカライズ指示	SMint 変換後の関数コール	
	並列セクション開始時	並列セクション終了時
copy	sms_copyin_array()	sms_reflect()
copyin / scopyin	sms_copyin_array()	sms_discard()
create / screate	sms_create_array()	sms_discard()
copyout	sms_create_array()	sms_reflect()

```
#pragma SMint parallel copyin(A[s1:n1][]) create(B[s2:n2][]) copyout (C[s3:n3][])
{
  :ノード並列処理

}//parallel section
```

```
{
 // shared配列宣言のサイズ          部分配列サイズ      データID
int __glb_A[2]={NY,NX}; int __size_A[2]={n1,NX}; int __id_A;
int __glb_B[2]={NY,NX}; int __size_B[2]={n2,NX}; int __id_B;
int __glb_C[2]={NY,NX}; int __size_C[2]={n3,NX}; int __id_C;

sms_copyin_array ( A[s1][0], __glb_A, __size_A, 2, sizeof(double), &__id_A);
sms_create_array ( B[s2][0], __glb_B, __size_B, 2, sizeof(double), &__id_B);
sms_create_array ( C[s3][0], __glb_C, __size_C, 2, sizeof(double), &__id_C);

   :    ノード並列処理

} //parallel section

sms_discard( &__id_A); // キャッシュデータの廃棄
sms_discard( &__id_B); // キャッシュデータの廃棄
sms_reflect ( &__id_C); //キャッシュデータの反映
sms_sync();    //実行同期
```

図 13 SMint における並列セクションの
データローカライズの変換例

4. おわりに

筆者らは，ソフトウエア分散共有メモリ mSMS を利用し，クラスタ上に共有メモリプログラミングモデル・PGAS モデルを基本とするマルチノード並列 C プログラミング環境を実現してきた. 本報告では, 典型的な並列ループ処理を，逐次プログラムから容易に変換することが可能なディレクティブベースの SMint API を提案し，並列処理部分の前後であらかじめアクセスするデータをローカライズする指示句を新たに導入した．これに伴い，データ毎に並列処理部の開始時と終了時の挙動を別々に指定できるようにし，ノード間データ転送と並列処理を効率化した．一方，アクセスデータが予めわからない応用などでは，これまで同様，segv ハンドラによる遠隔データフェッチを用いることができる．また，プログラマや応用に応じて，C プログラム，MpC プログラム，SMint 利用の 3 種のプログラミング環境が選択可能になる．

現在，提案した SMint を用いたステンシル計算のプログラム記述から C プログラムへの変換が可能となっている．対応する SMS 関数の実装後，各応用における実行性能を評価するとともに，様々な応用例についても，記述性を検討し，改良・追加していく予定である．

参考文献

[1] M.D. Wael, et al.: “Partitioned Global Address Space Languages”, Journal of ACM Computing Surveys (CSUR), Vol.47, No.62 (2015)
[2] 緑川博子, 飯塚肇 :"ユーザレベル・ソフトウエア分散共有メモリ SMS の設計と実装", 情報処理学会論文誌ハイパフォーマンスコンピューティングシステム,Vol.42, No.SIG9(HPS 3), pp.170-190, (2001,8)
[3] 緑川博子, 飯塚肇:"メタプロセスモデルに基づくポータブルな並列プログラミングインターフェース MpC",情報処理学会論文誌：コンピューティングシステム,Vol.46 No.SIG4(ACS9), pp.69-85,（2005,3）
[4] 直木三華, 緑川博子,甲斐宗徳："マルチコアプログラムにノード並列機能を加える API の提案", 第 11 回情報科学技術フォーラム FIT 論文集(第一分冊),B-003, pp169 -171（2012,9）
[5] 緑川博子,岩井田匡俊："マルチスレッド対応型分散共有メモリシステムの設計と実装", ハイパフォーマンスコンピューティングと計算科学シンポジウム HPCS2015, HPCS2015 論文集,（2015,5-19）
[6] Tsubame3 http://www.gsic.titech.ac.jp/tsubame3
[7] 緑川 博子: "ソフトウエア分散共有メモリシステム mSMS による大規模マルチコアノードにおけるステンシル計算", 情報処理学会, 研究報告ハイパフォーマンスコンピューティング（HPC）,Vol.2018-HPC-165, No.22, pp.1-8 (2018-07-31)
[8] OpenMP https://www.openmp.org/
[9] OpenMP 5.0 仕様書 (2018.11) https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf
[10] OpenACC https://www.openacc.org/specification
[11] OpenACC 2.7 仕様書（2018.11） https://www.openacc.org/sites/default/files/inline-files/OpenACC.2.7.pdf
[12] Berkely UPC http://upc.lbl.gov/ ver.2.28.9(2018.7.20)
[13] “Tarek el-ghazasi, et al. “UPC Distributed Shared Memory Programming”, WILEY, 2005, ISBN-10-471-22048-5
[14] Ruud van der Pas, et al. “Using OpenMP—The Next Step: Affinity, Accelerators, Tasking, and SIMD”, The MIT Press, 2017/10/20
[15] Xcalable MP http://www.xcalablemp.org/ja/
[16] XcalableMP 1.4 仕様書 (2018.11) http://www.xcalablemp.org/download/spec/xmp-spec-1.4.pdf